ActiveMQ

- ActiveMQ Authentication and Setup
- ActiveMQ 101
- ActiveMQ in Spring

ActiveMQ Authentication and Setup

Authentication

The particular Docker image of ActiveMQ that I'm currently using doesn't support authentication by default, you will need to enable the authentication plugin in order to enforce some kind of authentication.

To do so you will need to add the following plugin into the `activemq.xml` file and under the `broker` section tag.

activemq.xml

```
    <plugins>
       <simpleAuthenticationPlugin anonymousAccessAllowed="false">
         <users>
             <authenticationUser username="system" password="system"
groups="users,admins"/>
             <authenticationUser username="test01" password="test01"
groups="users,admins"/>
         </users>
       </simpleAuthenticationPlugin>
    </plugins>
```

You can create your own Docker image and copy this particular file overriding the default one that's in the container.

An example of Docker image and Docker compose file:

Docker image

```
FROM symptoma/activemq:5.15.13


COPY ./activemq.xml /opt/apache-activemq-5.15.13/conf/activemq.xml
```

Docker compose

```
services:
  amq:
    build: .
    ports:
      - 61616:61616
      - 8161:8161
```

Now you will actually have to provide a valid username and password in order to access to the broker, without this you can access the broker without any authentication.

To set the username and password with Spring Boot JMS you can create a ConnectionFactory bean like below

```
@Bean
public ConnectionFactory connetionFactory() {
    var factory = new ActiveMQConnectionFactory();
    factory.setUserName("system");
    factory.setPassword("system");
    factory.setBrokerURL("tcp://localhost:61616");
    return factory;
}
```

Then you can do a constructor injection of `JmsTemplate` object which gives you access to send and receive messages from the ActiveMQ.

> For whatever reason if you don't configure your own ConnectionFactory and instead use the default auto-configured ConnectionFactory the Spring boot application will actually not terminate immediately after your Spring Boot application finishes. Perhaps this isn't a problem if your application is web application, but for those application that just runs and finishes you will want to consider the option of configuring your own ConnectionFactory.

ActiveMQ 101

Background

To talk about ActiveMQ we need to talk about JMS (Java Messaging Serivce). It is an API **specification** defining the contracts between senders and receivers on how to create, send, receive, and read messages, but it doesn't give you the **implementation**, how you exactly do them. It is left to the JMS provider to define them.

ActiveMQ is one of the JMS provider that adhere to the JMS specification and allows you to use the API to communicate between the message broker.

There are other JMS provider:

1. ActiveMQ
2. RabbitMQ - Plugin is required
3. JBoss MQ
4. Hornet
5. IBM Websphere MQ

ActiveMQ

ActiveMQ is a message broker that decouples the clients with the servers that wants to communicates with each other. It can hold messages inside the queue until the server have capacity to operate on the messages that the clients have sent over.

It can operate both as a normal queue or publisher-subscriber model (Topics).

We will go over both type of operations.

In addition, ActiveMQ supports many protocols

- JMS
- Stomp -> This protocol allows you to use stomp.py to use Python as the interface for writing messages to ActiveMQ
- AMQP
- MQTT

Normal Queue

As a normal queue service, it does what it sound like, first-in, first-out message retrieval. You will have a producer to the queue pushing messages into it, and you will have consumers on the other side of the queue polling messages and actively consuming from it.

Creates a point to point communication system.

Each message that are received by the queue are then load balanced across consumers. **Meaning one message one consumer**. The queue delivers the message to a consumer and the consumer then processes the message doing whatever it needs to do with the message and finally acknowledges the message. After the message has been acknowledged it will then disappear from the queue and will not be available to be delivered again.

If the acknowledgement is not received by the server, either server crashed before the acknowledgement is received or the consumer went down and couldn't acknowledge it after the server sent out the message, that particular message will be available to be delivered to a consumer again.

Topic

Creates a one to many communication system. You have publisher sending messages to ActiveMQ, and the recipients who are subscribed to the topic will all receive the broadcast of the same message.

For each topic, you can have multiple subscription (consumers) to the topic, every subscription will get a copy of the message that are sent to the topic, very different than queue as each messages are received by only a single consumer.

Durable subscription

A subscription can also be optionally durable meaning that the messages sent to the topic will be kept until the subscriber consumes them. **Durable subscription is configured per consumer**, so that each consumer will get their own copy of the message durably.

For example, if we have a durable subscriber S that subscribed to a topic T at time D1. If a publisher starts sending messages M1, M2, M3 to the topic. S will receive all three of the messages. Then S is stopped but publisher continues to send M4 and M5.

When S is restarted at D2, publisher sends M6 and M7. Now S because it is durable subscribers it will receive M4 and M5 that were sent before it crashed follow by M6 and M7 and all future messages.

If the subscription weren't durable it will only have received M1, M2, M3, M6 and M7, missing out on M4 and M5.

Non-durable subscription can only receive messages that are sent out during the time that the subscription is alive.

To identify a non-durable and durable subscription from each other, during the creation of the durable subscription you must specify a client id and the name of the durable subscriber name. This is so that it can identify which message for which subscription to keep around if it is for a durable subscriber or not.

Important: For durable subscriptions it works similarly to a queue in the sense that you get the guarantee of delivery. It will retain all the messages from the topic's publishers until they are delivered to, AND acknowledged by a consumer of the durable subscription. Source

If for whatever reason the durable subscription consumer can't acknowledge the message because it is down, the message will be kept and reattempted when the consumer is back online, until the ack is received by the server. This is PER durable subscription, so if you have multiple durable subscription setup, each of them will be keeping track for their consumer the messages.

You must create the durable subscription BEFORE the messages are attempted, otherwise, if no subscription exist for the topic, the messages will be gone.

For non-durable subscriptions if you don't acknowledge the message well, it doesn't really make a difference because it is the nature of non-durable subscriptions. The ActiveMQ will not resend the messages to the non-durable subscriptions.

Persistent and non-persistent messages

When a message is configured to be persistent, the message broker will store the message in a store on the disk to be recovered later if the broker goes down or is shutdown and later restarted. This is for resiliency and messages will survive server failures or restart.

If a message is sent as non-persistent then it is only store in memory therefore, if the broker is stopped for some reason then those non-persistent messages are then lost for good.

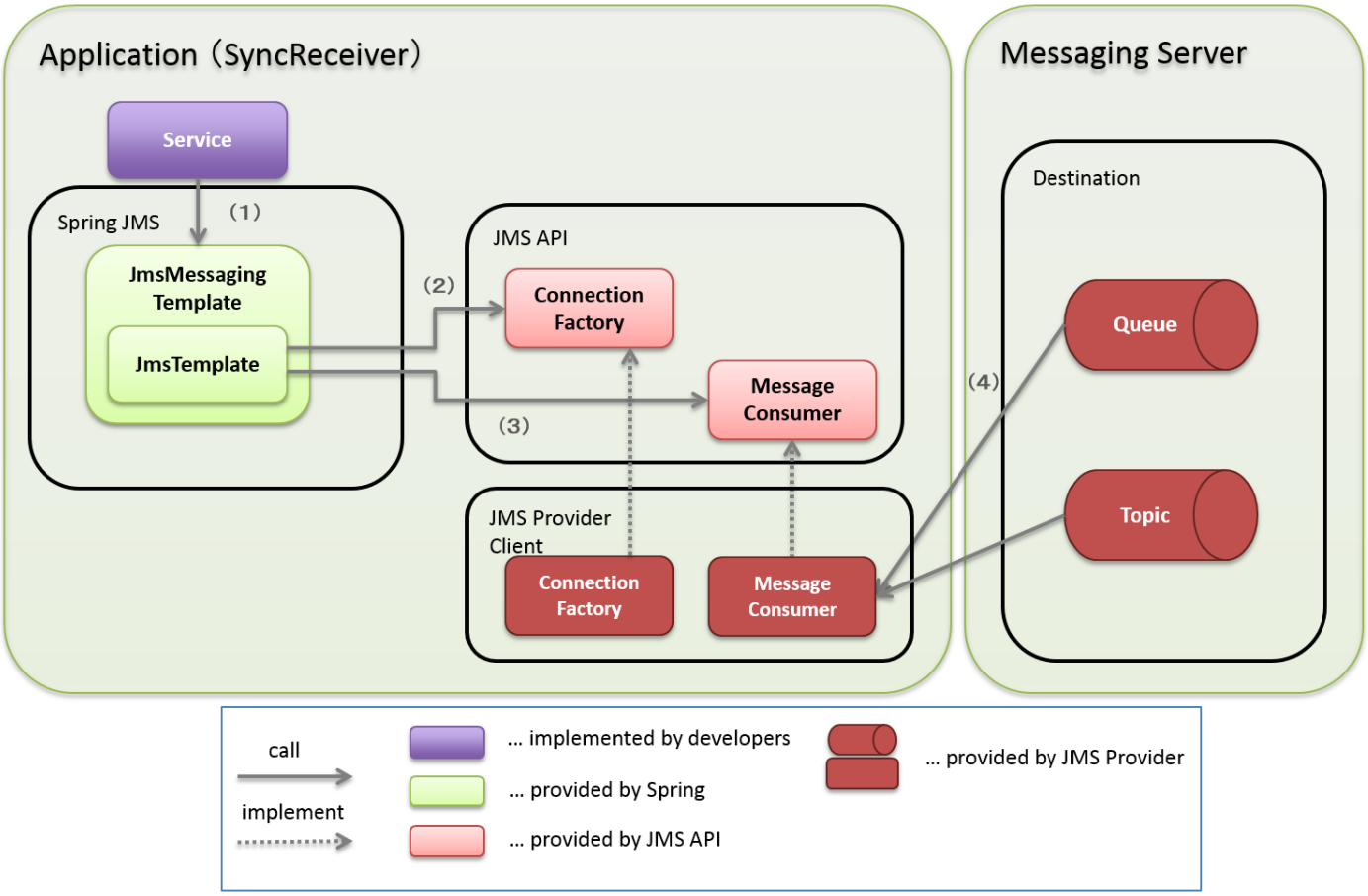
This persistent behavior applies to both queue and topic because they are per message configuration.

The default delivery mode is persistent unless you configure it to not be persistent.

Making persistent messaging is usually slower than non-persistent delivery because there is no need to write the file into disk, but you get the guarantee that the message will still stay if in case the message broker went down.

ActiveMQ in Spring

Using ActiveMQ in Spring



I just want to put this diagram here so that this can be referenced if needed in the future. If you are planning on using Spring to interact with ActiveMQ, then you will be using something called `JmsTemplate`. This class essentially simplifies the interaction with the JMS API (behind the scene there is resource adapter that eventually calls the ActiveMQ driver call to connect to the ActiveMQ instance from the JMS API Source).

You just need to worry about the JMS API which you can use to send and receive messages from the message broker easily, without actually invoking the ActiveMQ classes. It is unified into one interface for you to interact with many different kind of JMS providers.

> I will only be going over how to use `JmsTemplate` because there is really no need to interactive with the actual ActiveMQ implementations. Unless you're curious...